Tampere University

Nayeong Song

Comparison between lazy and strict evaluation

Bachelor's thesis
Faculty of Information Technology and Communication Sciences (ITC)
Examiner: Maarit Harsu
Oct 2020

ABSTRACT

Nayeong Song: Comparison between lazy and strict evaluation
Bachelor's thesis
Tampere University
Bachelor's Degree Programme in Science and Engineering
Oct 2020

Evaluation strategy is the way to define the order of function call's argument's evaluation. Most of the today's programming languages employ strict programming paradigm, which is focused mostly on call-by-value and call-by-reference. However, call-by-name or call-by-need, which evaluates the expression lazily, is used only in functional programming languages and those are rather unfamiliar to these day's programmers.

In this thesis, the focus is on the comparison between different evaluation techniques but mostly focused on strict evaluation and lazy evaluation. First, the theoretical difference is presented in mathematical aspect in lambda calculus, and the practical examples are presented in Python and Haskell.

The results show that a program can benefit lazy evaluation when evaluating non-terminable expressions or optimize the performance when the values are cached. However, in case the function has side effects, lazy evaluation cannot be used together and it makes harder to predict the memory usage compared to strict evaluation.

Keywords: evaluation strategy, lambda calculus, algorithm efficiency, strict evaluation, lazy evaluation, Haskell, Python

PREFACE

This thesis was written in Tampere, Finland between September 2020 and October 2020. The idea of the topic was inspired from the course Principles of programming languages. The course introduced different expression evaluation strategies briefly and I wanted to do research on it more deeply. I sincerely express my gratitude to my supervisor, Maarit Harsu, for her advide and guidance and support throughout this thesis writing.

Tampere, 5.10.2020

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

Definitions of efficient code include using appropriate algorithms or avoiding unnecessary steps or variables. However, the evaluation strategy is seldom mentioned along with the coding efficiency, or the mathematical knowledge behind it is rarely cared for by programmers. How the programming expressions are evaluated can significantly impact on the quality of the program and the characteristics of the language.

Most of the modern languages accept strict evaluation techniques. For strictly evaluated languages, when a compiler encounters a programming expression, it is immediately evaluated. In contrast, lazy evaluation delays the expression until it is needed in a program, and it is usually used in functional programming languages. To understand the functional programming, lambda calculus is often accompanied, which expresses the computation on function abstraction and application using lambda terms.

The goal of this thesis is to understand the fundamental differences between strict evaluation and lazy evaluation in the lambda calculus aspect and study on which expression evaluation method is beneficial in various situations.

This thesis will present different evaluation strategies, the principles behind it in lambda calculus, and how they are implemented in actual programming languages.

Chapters 2 and 3 briefly explain what is programming expression and how is algorithm efficiency defined. In Chapter 4, the basics of lambda calculus are introduced to give the basis for further explanation of reduction orders. After theoretical comparisons for different evaluation methods are made in Chapter 5, some practical examples of algorithm efficiency in one strict language Python, and one non-strict language Haskell are reported in Chapter 6. Discussion about results are made in Chapter 7 and finally in Chapter 8, all the topics introduced in this paper are recapped.

2 EXPRESSIONS

Typically, expression refers to the syntactic entity that can be reduced into value. Unlike the mathematical expressions that consist of only numbers and arithmetic operators, expressions in programming language include function calls, identifiers such as variable names, and assignment operators. This combined expression can be reduced into one of the primitive data types, and this is called evaluation of the expression. Here, primitive data types include character, integer, floating-data point, fixed-data number, boolean, or reference. [12]

2.1 Abstract Syntax tree

Programming languages indicate computations in a way that is comprehensible to both human and machine. The syntax of the language defines a way in which phrases (expressions, commands, declarations) are combined to form programs and this can be defined into two: concrete syntax, and abstract syntax. The concrete syntax (external representation) is defined by strings or values generated by the grammar. This defines the way the program looks to the programmer and the way that expression looks like. However, such data cannot be directly processed in the computer, but it should rather be converted into the abstract syntax (an internal representation) which represents the significant parts of the expression.

Abstract syntax tree (AST) gives a convenient way to visualize the internal representation. To create an AST for certain concrete syntax, each production of the concrete syntax is named and associated with each internal node of the tree. For the edges of AST, it can be labeled with the corresponding names of non-terminal occurrence and the leaves correspond to terminal strings.

Having such a distinction between concrete syntax, which is readable by human, and abstract syntax, which is readable by computers, conversion from one to another should be considered as well. The conversion rule can differ depending on how concrete syntax looks. For example, if the concrete syntax is a group of strings or characters, deriving an abstract syntax tree from concrete syntax may be complex. In this case, the task done is called *parsing* which is done by *parser*. A *parser*

generates a parse tree which shows the concrete syntax (how tokens are grouped together). Parse tree consists of nodes (non-terminals, syntactic categories) and leaves (terminals, tokens). In contrast, abstract syntax tree consists of nodes (constructor functions) and leaves (atoms, zero-place constructor functions).

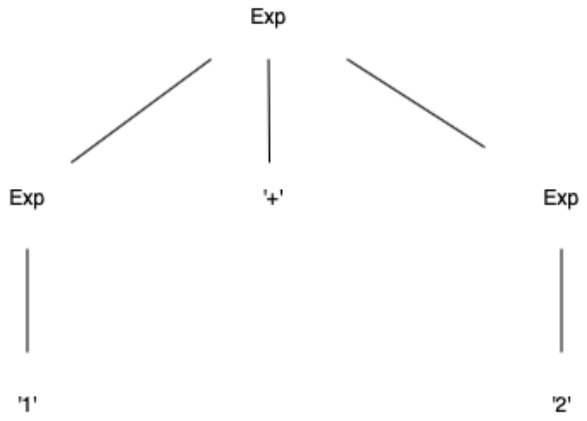

Figure 2.1 *Parse tree example*

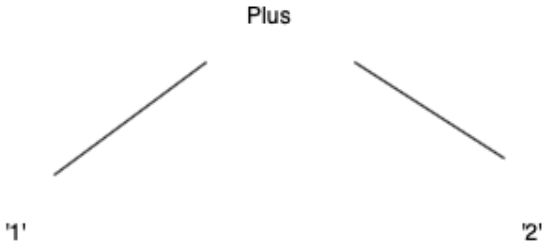

Figure 2.2 *Abstract syntax tree example*

The Figure 2.1 and Figure 2.2 gives brief examples of parse tree and abstract syntax tree for the same expression 1 + 2. Typically, when converting a parse tree into an abstract syntax tree, the rules are labeled (Plus in this example), terminals are ignored, and labels are treated as a constructor name.

2.2 Expression evaluation

For evaluating an expression, the language processing system in the computer should be explained first. Firstly, the raw text input in a source language is passed to a front-end which converts the text into an abstract syntax tree. At front-end, a series of characters are converted (grouped) into meaningful units. This grouping process

can be divided into two parts: scanning and parsing. Scanning is a process for dividing the sequence of characters into a token, and parsing is a process that organizes the sequence of tokens into hierarchical syntactic structures such as expressions. After this, in an interpreted language, the syntax tree is passed as in input to the interpreter. In a compiled language, the difference is that the interpreter is replaced by a compiler, which converts the abstract syntax tree into a translated program to be compiled in some other languages such as assembly, machine language, or lower-level language, and this can be executed in the target language. The below graph shows the block diagrams for a language processing system. [2]

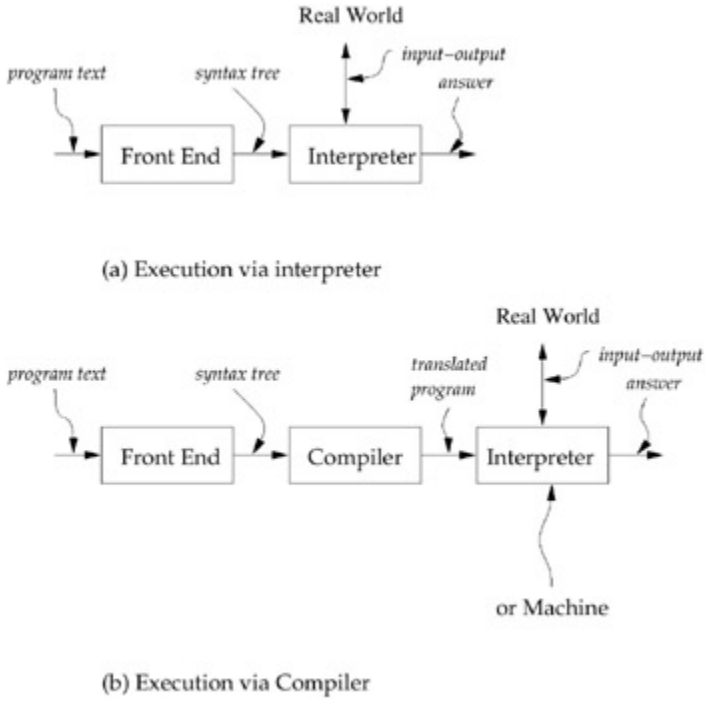

Figure 2.3 *Block diagrams for a language processing system [2]*

Regardless of the implementation strategy, front-end is needed for converting programs into abstract syntax trees since programs are simply composed of strings and characters. Front-end is responsible for grouping these characters into meaningful units.

3 ALGORITHM EFFICIENCY

An algorithm is a sequence of instructions for obtaining the expected output for any legitimate input in a finite amount of time. Algorithm efficiency is mainly defined in two areas: space and time complexity. Typically, the less time and less space the better algorithm, but this is not always possible, so rather the compromise between them should be made. Both time and space complexity are functions of the size of the problem to be solved, and the larger the problem, the longer it takes in the algorithm.

Space (memory space) complexity measures of the algorithm's working storage requirement. For big data or embedded systems programming, space complexity analysis is essential, and components of space/memory use can be classified as three big categories: instruction space, data space, and run-time stack space. [8]

Measuring the exact time(run time) for executing the program is hard since it depends on many factors. As a first-hand approach, just running the program and measuring the running time will result in different outputs depending on the computer. Rather, time efficiency means merely an indicator of the amount of work the algorithm has to complete compared to the size of the problem. By analysis of the code, simply operation counts or step counts of the code can be estimated as time efficiency whereas by the execution of the code, some benchmark which applies various data set to measure the performance can be used. The universal measure time defines it by computing the number of elementary actions carried out by the processor when executing the algorithm, which depends on neither computer nor programming language. [3]

Typically computational power of modern computers does not make many problems in dealing with time complexity. However, as the system size increases, when both space and time complexities increase, the advantage of one algorithm to others is more dominant. Thus, usually, complexity analysis is considered when the system structure's size is very large. [8]

4 LAMBDA CALCULUS

Lambda calculus and combinatory logic were invented in the 1930s by Alonzo Church at Princeton University as a mathematical system for defining computable functions. Its main goal was to describe the basic properties of function abstraction, application, and substitution in a general way. Although lambda calculus is highlighted as a branch of mathematics, it leads to considerable impact in the field of programming as well. [7]

The syntax of lambda calculus follows the recursive rules using the variables, parenthesis, spaces, and the symbol λ. The lambda calculus is a fundamental theoretical building block for functional programming languages and has various applications in artificial intelligence, and logic. With its simplicity, lambda calculus has been used for the analysis of programming languages. [6]

4.1 Lambda terms

In pure lambda calculus, there are three kinds of terms: variable, abstraction, and application. Lambda calculus consists of combining lambda terms and applying various reduction rules on them to evaluate the expression.

- a variable x itself
- $\lambda x.e$: abstractions
- $e_1\ e_2$: applications

Abstractions typically serve as functions, and applications represent the application of a function to its argument. The binding of λ is extended as far right as possible, and free variable in lambda calculus means that the variable is not bound by a lambda. For example, in function $\lambda x.x$, x variable is not free but in function $\lambda y.x$, x is a free variable. Besides, "closed" lambda terms means that there are no free variables. If there are free variables left, then lambda term is called "open". [6]

In programming languages, the lambda function (anonymous) is an example where the term is derived from lambda calculus. Following the rules of lambda calculus, a function (abstraction) does not need to be named and provides a nice way to write closures.

For example, a standard Python function is defined using the keyword def with the specification of the name of the function.

```
def function(x):
    return x
```

In contrast, using lambda construction in Python allows the programmer to create a function without specifying a function name (anonymous function) and the structure of the lambda function follows the fundamental structures of abstraction in lambda calculus.

```
lambda x: x
```

Besides, typically the term *free variable* in a programming language is used in the same way as in lambda expressions. If the variable is not bound and referenced in the body of the expression, it is called a free variable.

4.2 Reduction

Like any programming expressions, lambda expression has its meaning that results in its all function applications (combinations). Evaluating a lambda expression is called reduction, and this involves substituting free variables in a similar way how formal parameters are substituted by actual parameters. [7] Mainly, the evaluation of a lambda expression consists of a series of β-reduction rule, which is applied until no more reduction rules can be done.

Formally, β-reduction replaces bound variable in a function body with a function argument.

$$(\lambda x.e_1)e_2 \longrightarrow e_1[e_2|x] \tag{4.1}$$

Here, notation $e_1[e_2|x]$ means substituting all instances of e_2 in e_1 by x. And the syntax $\longrightarrow$ is used as a shorthand for beta reduction. That is, the equation 4.1

means that the beta reduction of $(\lambda x.e_1)e_2$ is $e_1[e_2|x]$. So the beta reduction removes the symbol λ, and the function's body with the argument x is resolved. [6]

A β-reduction expression (β-redex) is an expression to which β-reduction can be immediately applied. In other words, β-reduction expression is a lambda expression in which the first term is a function abstraction.

The purpose of simplifying a lambda expression is to evaluate the value of it. A lambda expression is called a normal form if it does not contain β-reduction expressions so that it cannot be further reduced using a β-reduction rule. As β-redex is the expression where β-reduction can be directly applied, a normal form has no more function applications to evaluate. So the reduction strategy should guarantee that normal form is produced as a result. [7] Besides, according to Church-Rosser Theorem, for a lambda expression that has more than one way to reduce the lambda expressions, they will result in the same normal expression. However, this theorem does not guarantee the existence or validity of the answer. [1]

5 EVALUATION ORDER

The evaluation strategy is a set of rules for evaluating expressions in a programming language. Evaluating expression means the evaluation of the arguments of a function call and passing value of the function. However, in the lambda calculus perspective, it can be viewed as rewriting the expression in a simpler format, which eventually results in normal expression. Two important orders or rewriting a lambda expression can be defined: normal order, applicative order. Normal-order reduction chooses the left-most β-redex first to evaluate whereas applicative-order reduction chooses the right-most β-redex first. [6]

Would it matter which reduction strategy we use? A reduction strategy affects both performance (how many reduction steps are required to reach a normal form) and termination (whether the normal form can be achieved or not).

5.1 Normal order reduction

Normal order reduction reduces the leftmost β-redex first before reducing the sub-expressions inside of it and those that follow it. In other words, the lambda expression's leftmost occurrence of a function application is rewritten, and this evaluation method is also known as call-by-name. Since it delays its evaluation of the arguments to a function, any expression for which termination is possible will be terminated by this reduction method whereas applicative order reduction may not. In terms of performance (number of evaluation steps), normal order evaluates the argument as much as the number of times it is used in a function body. [13]

However, typical lazy evaluated languages do not use call-by-name evaluation since it requires too expensive computation in practice, but call-by-need. The reason is that β-rule may copy the argument as much as it is called whereas the call-by-need guarantees that the argument to a function is not copied before it is reduced to a value. In call-by-need, any terms can be discarded if it is not needed and the term should not be duplicated until it has been reduced to a value. [9] Call-by-need can be viewed as an optimized (memoized) version of call-by-name since it stores the results of expensive function calls and returns the cached result when the function

with the same input occurs again. The most common reason for using memoization is to avoid the repetition of sub-calculations within a function. [10]

Some languages such as Scheme provides this feature in the form of delay and force, in which arguments are passed unevaluated. These functions provide the implementation of lazy evaluation. Lazy evaluation has same semantics as normal-order evaluation, but it uses memoization, which caches the expressions that are already calculated so that it can be reused if they are called again. Lazy evaluation can be either default in a language or can be simulated in strictly evaluated languages.

5.2 Applicative order reduction

Applicative order reduction evaluates its leftmost innermost redex first. This means that function's arguments are evaluated before the function is applied. So the innermost expression must be reduced before it is substituted into the body of the function that contains it as an argument. Known as call-by-value (eager evaluation), the body of the function is not evaluated until the function is called. [14]

However, applicative order may fail to terminate the evaluation if a function given is a non-terminating expression as an argument while normal order may succeed in evaluating non-terminable expressions. In this case, applicative order may cause the expression evaluation process to loop forever. In terms of performance (number of evaluation steps), applicative-order reduction evaluates all constituent expressions, some of which are unnecessary. When this happens, there is often a need to control the evaluation process by defining the special condition. [11]

Generally, in respect of clarity, applicative-order is preferred over normal-order evaluation. Functions in C, C++, Java, and Scheme including many other languages follow applicative order reduction, which means that all the actual parameters are evaluated and bound to the formal parameters at a call site. Within most current programming languages, parameters are passed by value by default with the arguments as a copy of the calling value.

5.3 Comparison between Normal order and Applicative order

Following the definitions of normal order and applicative order, the main difference between normal order reduction and applicative order reduction lies in when the arguments are evaluated. Applicative order evaluates arguments before the procedure is called whereas normal order delays evaluation of arguments until it is

necessary. To be specific, normal order passes a representation of the unevaluated arguments to the subroutine instead and evaluate them only if needed. However, the fact that arguments are not evaluated first may cause slowing down normal order reduction than applicative order reduction. As for the value for the lambda expression, using different evaluation methods will result in the same value. According to Church-Rosser Theorem, if there are no side effects or mutation (expressions can be repeatedly evaluated without any other effects), two evaluation methods should give the same results but may end up evaluating the same term more times than the other method. [11] For example, below pseudo-code shows the function square which is a function that returns the squared value of an input number.

```
define (square x) (* x x)
```

In normal order evaluation, the leftmost outermost reducible expression is evaluated first at each step. In the example below, innermost square function's argument's evaluation is delayed as much as it can.

```
square (square 3) =>
* (square 3) (square 3) =>
* (* 3 3) (* 3 3) =>
* 9 9 =>
81
```

In applicative order, the leftmost innermost reducible expression is evaluated first at each step. When applying the square function twice, argument of the outermost square function, which is inner square function's argument is evaluated first.

```
square (square 3) =>
square (* 3 3) =>
square 9 =>
* 9 9 =>
81
```

The most notable difference between two is the number of times that multiplication operator (*) is applied. In applicative order, multiplication operator (*) was applied twice whereas it was applied three times in normal order. However, in lazy languages which use call-by-need method, the results of the first evaluation in normal order reduction would be cached so that the same expression is not evaluated twice.

While normal order evaluation may result in extra steps to be evaluated, applicative order may result in problem that do not terminate when the normal-order evaluation can. Below example shows the case when normal order fails to

```
if True x y = x
if False x y = y
define (greater x y) = x > y
define (divide x y) = x/y
```

In normal order reduction, leftmost outermost reducible expression is evaluated recursively, which results in the value.

```
if (greater 6 3) (square 3) (divide 1 0) =>
if True (square 3) (divide 1 0) =>
(square 3) =>
* 3 3 =>
9
```

However, in applicative order, it results in run time error when evaluating (divide 1 0) since dividing by 0 is not possible.

```
if (greater 6 3) (square 3) (divide 1 0) =>
if True (square 3) (divide 1 0) =>
if True (* 3 3) (divide 1 0) =>
if True 9 (divide 1 0) =>
if True 9 (1/0) =>
valueError: Divide by zero
```

6 PRACTICAL COMPARISON OF THE EVALUATION METHODS

The focus of this thesis is to study the theoretical/ practical differences in expression evaluation methods. However, the main reason for comparing different expression evaluation methods is to determine which situation is more beneficial to use either evaluation method. In this section, we identify the practical difference between normal-order and applicative-order evaluation methods in terms of algorithm efficiency. However, as proved in the evaluation order section, languages that use normal-order evaluation without memoization are by no means common because of duplicated evaluation of the same expression. That is why most of the normal-order evaluated languages accompany memoization, which saves the expression value so that the same expression is not evaluated again. Thus, our method compares the algorithm efficiency in lazy evaluation (memoized normal-order evaluation) and strict evaluation (applicative order evaluation). To show the existence of lazy evaluation in strict languages, Python was chosen as an example, and to show the features of functional programming languages which typically accepts lazy evaluation by default, Haskell was chosen. Several data structures and functions were presented in both languages and both algorithmic efficiency and the logic of each evaluation method was simulated in code.

6.1 Comparison arrangements

In Python 3, function range() which simulates laziness, was chosen as an indicator of a lazy iterable. In addition, list comprehensions and generators were compared to show the performance difference.

As of space complexity, the module Memory Profiler was used for monitoring memory consumption of the each code line. The line-by-line memory usage enables us to analyze the incremented amount of data usage per step so that we can compare the memory usage between lazy evaluation and strict evaluation.

The time complexity was measured using the module time. Stating and ending

points were set between the code we wish to analyze, and the execution time was calculated by subtracting one time from the other time.

In Haskell, arguments are not evaluated until they are passed to a function and the values are actually used (lazy by default language). This lazy evaluation allows to bypass undefined variable or infinite data, but this can also make some drawback that makes it hard to predict the memory allocation. In this paper, the function myFunction which accepts three integers as arguments and returns the integer as a result was implemented to visualize lazy evaluation.

The time and space profiling was made using GHC (Glasgow Haskell Compiler). The program is recompiled before generating a profile with the file named prog.prof where prog is the name of the program. The result from GHC tells the total time and total memory allocation measured during the run of the program. [5]

6.2 Results concerning Python

As a result of Python, creating function range in Python 2 and Python 3 gave different outputs. In Python 2, range function returned a list of numbers within the range whereas Python 3 returned a iterable object. range() in Python 3 is equivalent to xrange() in Python 2, and it accepts the lazy evaluation behavior.

```
>>> my_range = range(100)
>>> type(my_range)
<type 'list'>
```

```
>>> my_range = range(100)
>>> type(my_range)
<class 'range'>
```

Another example of lazy evaluation in Python is generator. A generator does the same thing as what list comprehension does, but the difference between them is quite similar as range() in Python 2 and 3: lazy evaluation. A list comprehension immediately returns an actual list just as range() does in Python 2. In contrast, a generator expression returns the object that is iterable.

```
>>> list = [x for x in range(10)]
>>> print(list)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> generator = (x for x in range(10))
>>> print(generator)
<generator object <genexpr> at 0x100774660>


lis = [x for x in range(500000)]
gen = (x for x in range(500000))
```

	memory usage (MiB)	time (sec)
List Comprehension	0.6992	14.51
Generator	0.0039	7.01

Table 6.1 *Memory usage and time profiling for list comprehension and generator*

Table 6.1 shows the result of memory usage and time for using list comprehension and generator. The generator yields one item at a time and item is generated only when it is needed whereas list comprehension reserves memory for the whole list. Thus, generator is more memory efficient when handling a lot of data and avoid keeping all unnecessary elements in memory. However, generator can be iterated once at a time, so in case the program needs to iterate multiple times, generator must be recreated. Besides, if the final goal is to generate another list, then generator is not suitable.

6.3 Results concering Haskell

To test how lazy evaluation affects performance in Haskell, myFunction (Figure 6.2) which accepts three integer variables a,b,c as arguments and returns the integer was implemented. Inside the function myFunction (Figure 6.2), another function double (Figure 6.1) which accepts two integers a, b and returns the sum of those numbers are included as well.

```
1 double :: Int -> Int -> Int
2 double a b = a + b
```

Figure 6.1 *Haskell code for the function double*

```
1  myFunction :: Int -> Int -> Int -> Int
2  myFunction a b c =
3    let result1 = double a b
4        result2 = double b c
5        result3 = double a c
6    in if result1 < 100
7         then result1
8         else if result2 < 100
9           then result2
10          else result3
```

Figure 6.2 *Haskell code for the function myFunction*

In typical strict languages which accept applicative order evaluation methods, series of commands are executed in order. In those languages, the function myFunction in Figure 6.2 clearly shows inefficiency since the variables result1, result2, and result3 are evaluated immediately after they defined. However, in Haskell the computation for the variables result1, result2, and result3 are delayed until they are actually needed.

To clarify how lazy evaluation actually works and how the memory, time profile changes depending on the order of computation, the sum of three lists were passed as parameters for the function myFunction to make the computation more expensive than addition (function double) itself.

Firstly, the sum of small lists were passed as first and second parameters of myFunction and the last parameter was set as the sum of infinite list, which is theoretically not defined. And as a second case, the order of arguments were exchanged but set the limit for infinite list.

```
Case1: myFunction (sum [1..5]) (sum [1..5]) (sum [1..])
Case2: myFunction (sum [1..9999999]) (sum [1..5]) (sum [1..5])
```

	memory allocation (byte)	time (sec)
Case 1	55504	0.00
Case 2	1280055720	0.49

Table 6.2 *Memory usage and time profiling for Case 1 and Case 2*

As a result (Table 6.2), the running time for case 1 was nearly zero even though the last argument was infinite data structure. This is because when the result1 was calculated and satisfy the if statement that requires the value to be less than 100 (line 6 of the Figure 6.2), it immediately exits the function and does not evaluate result2 and result3. However, when the order of arguments were changed so that the first argument requires expensive computation, the memory allocation and time spent was much larger than case 1 even though the data were still same or smaller than the case 1.

Lazy evalution follows the principles of normal order reduction, but it is often combined with memoization, which stores a value of a function to prevent re-computation. To visualize the algorithmic efficiency more dynamically, Fibonacci recursion was implemented with and without memoization (see Figure 6.3 and Figure 6.4).

```
1 slow_fib :: Int -> Integer
2 slow_fib 0 = 0
3 slow_fib 1 = 1
4 slow_fib n = slow_fib (n-2) + slow_fib (n-1)
```

Figure 6.3 *Haskell code for the function slow Fibonacci without memoization [4]*

```
1 memoized_fib :: Int -> Integer
2 memoized_fib = (map fib [0 ..] !!)
3     where fib 0 = 0
4           fib 1 = 1
5           fib n = memoized_fib (n-2) + memoized_fib (n-1)
```

Figure 6.4 *Haskell code for the function Fibonacci with memoization [4]*

The results of Fibonacci function with and without memoization with two different inputs are shown in table below (see Table 6.3).

	memory allocation (byte)	time (sec)
slow_fib (n = 40)	25480935952	12.16
memoized_fib (n = 40)	61600	0.00
memoized_fib (n = 100000)	473907032	87.94

Table 6.3 *Memory usage and time profiling for Fibonacci with and without memoization*

For naive implementation of Fibonacci function, program already starts to slow down when n = 30, and the memory allocation is much bigger compared to using the memoized function. Memoized Fibonacci function managed to handle the computation for n = 100000 within around one and half minutes since memoization prevents calculating the same computation more than once.

While lazy evaluation allows us to save memory and enables us to formally define the infinite data structure, it has main drawback in memory management. Haskell's built-in function reverse (Figure 6.5) from the module Prelude creates a new string from the original in the reverse order.

```
1 reverse l =  rev l []
2   where
3     rev []     a = a
4     rev (x:xs) a = rev xs (x:a)
```

Figure 6.5 *Haskell code for the built-in function reverse [4]*

Inputting finite list into this function results in the list with the elements reversed from the input as below. However, what happens if the input is infinite list?

```
reverse [1..5]
>> [5,4,3,2,1]
```

```
reverse [1..]
```

Following the definition of reverse function, it continuously recurs itself until generating the empty list [ ]. However, in case the argument is an infinite list, this never happens, so the program will run forever.

7 DISCUSSION

In terms of performance, the results clearly show that the program can be optimized and reduce the computational complexity using lazy evaluation. Besides, lazy evaluation allows us to recursively define an infinite data structure and visualize it in code. Besides, as in normal-order reduction, lazy evaluation enables the expression that is not possible to terminate be computed by chance.

Examining the cases where lazy evaluation managed to reduce the time or space complexity in practice, one of the cases was when the evaluation occasionally avoid expensive computation. An example of the function myFunction in Haskell managed to evaluate the non-terminable expression only in some cases (when the third argument was infinite), but not in all cases. Avoiding unnecessary evaluation in code can be one of the ways to optimize the program, but rarely this is a major problem in the code.

Another example of the Fibonacci function in Haskell proved that code optimization can be made by caching the value to prevent duplicated computations, called memoization. Memoization becomes handy when a potentially huge range of input is expected but the range is still restricted and known. Besides, if we know that the program only uses a small subset of possible inputs, it can be useful as well.

Then why is the mainstream of the programming languages still focused only on strict languages although clear benefits are using lazy evaluation? There are several disadvantages.

Firstly, lazy evaluation cannot be used along with the functions with side effects. Usually, the functions that perform side effects require the code to be executed in a certain order. However, the lazy evaluation does not ensure that the code is executed in the right order if they are not needed, so the languages with non-strict evaluation must be purely functional to be useful. For the languages with strict evaluation, often macro or thunks should be accompanied to make the lazy (non-strict) functions to be useful. [11]

Secondly, hardware architectures in general are optimized for the languages with

strict evaluation. For the best compiler in the languages with non-strict evaluation, this produces slower code than the one with strict evaluation. In the programs which require very tight resource usage, direct access to the hardware is vital in such an environment, but depending on the functional programming languages with lazy evaluation, the programmer may or may not have this. [11]

Lastly, as shown in the example function `reverse` in Haskell, using lazy evaluation is harder to predict the memory usage and potentially lead to a memory leak. A memory leak occurs when a program allocates more memory than needed for the program. Haskell uses garbage collectors which deallocates unused allocated memory on occasions, but programmers should consider memory management in mind, and memory leak can slow down the garbage collector as well. There has been a substantial effort to deal with memory leaks in lazily evaluated languages by changing the compilation techniques or modifying the garbage collector, but it often produced disadvantages that can be against the benefits of the existing ones.

8 CONCLUSIONS

Most of today's programming languages accept strict evaluation for the programming expressions. Programming expression is a syntactic entity that can be reduced into a value, and how the expression is reduced (the order of reduction) varies between programming languages.

To understand the evaluation order on a deeper level, understanding lambda calculus is essential. Programming expression consists of lambda terms, which is a basic entry of the lambda calculus. According to the Church-Rosser Theorem, any expression which can be reduced results in the same value regardless of the order of the reduction. The reduction can be classified as two: normal-order reduction and applicative-order reduction.

Normal-order reduction reduces lambda expression's leftmost occurrence of a function application recursively, and it is known as call-by-name as well. In contrast, applicative-order reduces the leftmost innermost redex first, so the function's arguments are evaluated before the function is applied. However, normal-order reduction itself is not used in programming languages since it evaluates the same expression many times. Lazy evaluation in programming languages typically refers to normal-order reduction with memoization, call-by-need in other words.

Lazy evaluation can be either default in a certain language or can be simulated in strict languages. Haskell uses lazy evaluation by default, but the strict evaluation can be forced in Haskell as well. In this thesis, Python was used as an example of a strict language that can simulate lazy evaluation in some functions and Haskell as the language with lazy evaluation as a default.

In Python, list comprehension and generator were introduced as examples that do the same thing but accept different evaluation methods. The generator creates an iterable object in which the elements are consumed later and evaluated lazily whereas list comprehension creates the whole object immediately. In Haskell, it proved that lazy evaluation can be beneficial when it occasionally avoids the expensive calculation or uses caching which saves the value that is already evaluated.

However, lazy evaluation is not useful if it is used along with side effects. Functions with side effects usually have to follow some orders, but lazy evaluation may not execute the code in the correct order. Memory management can be another challenging point in lazy evaluation. Since it allows us to define the infinite data structure, memory cannot be predicted easily and the program fails to terminate or run out of memory when the expression evaluation is non-terminable.

The fact that functional languages are hard to understand is often brought as another reason why imperative programming languages are preferred, but this is rather subjective and this can be because today's programmers are used to imperative language and strict evaluation thinking.

Clearly, both strict and lazy evaluation has benefits and downsides, and neither of them is ideal. There are many things left to study in order to decide the evaluation strategies to optimize the program. In this thesis, lazy evaluation was mostly focused, but the characteristics of the strictly evaluated languages could have been dealt with more. Besides, lambda calculus was only briefly explained, but lambda calculus is a fundamental building block for functional languages and is crucial for understanding the evaluation order. The only reduction method was β-reduction, which is the most commonly used, but the other reduction strategies such as α, η-reduction could have been mentioned as well.

For the practical code examples, the next step could be to implement the memory management difference between the evaluation methods and identify why memory management is called hard in functional languages. And the theoretical comparison shows step by step how the reduction strategies are differently made in normal and applicative order, but in the practical part it was hard to show them but rather the time and space complexities were simply compared, so the better method for handling this can be suggested as well.

REFERENCES

[1] J. Alama and J. Korbmacher. "The Lambda Calculus". In: *The Stanford Encyclopedia of Philosophy*. Ed. by E. N. Zalta. Fall 2017. Metaphysics Research Lab, Stanford University, 2017.

[2] D. P. Friedman and M. Wand. *Essentials of Programming Languages*. 3rd ed. MIT, 2008. ISBN: 0-262-06279-8.

[3] J. Gal-Ezer and E. Zur. "The efficiency of algorithms—misconceptions". In: *Computers Education* 42.3 (2004), pp. 215 –226. ISSN: 0360-1315. DOI: https://doi.org/10.1016/j.compedu.2003.07.004. URL: http://www.sciencedirect.com/science/article/pii/S0360131503000848.

[4] HaskellWiki. *Memoization — HaskellWiki,* [Online; accessed 11-September-2020]. 2014. URL: https://wiki.haskell.org/index.php?title=Memoization&oldid=57978.

[5] HaskellWiki. *Performance — HaskellWiki,* [Online; accessed 11-September-2020]. 2018. URL: https://wiki.haskell.org/index.php?title=Performance&oldid=62590.

[6] P. Hudak. *A Brief and Informal Introduction to the Lambda Calculus*. YALE, 2008. URL: https://www.cs.yale.edu/homes/hudak/CS201S08/lambda.pdf.

[7] S. Ken. *Syntax and semantics of programming languages*, pp. 139–166. URL: http://homepage.divms.uiowa.edu/~slonnegr/plf/Book/.

[8] P. W. Kuo and M. J. Zuo. *Optimal reliability modeling: principles and applications*. Wiley, 2002. ISBN: 047139761X,9780471397618,9780471275459,047127545X.

[9] J. Maraist et al. "Call-by-name, call-by-value, call-by-need and the linear lambda calculus". In: *Theoretical Computer Science* 228.1 (1999), pp. 175 – 210. ISSN: 0304-3975. DOI: https://doi.org/10.1016/S0304-3975(98)00358-2. URL: http://www.sciencedirect.com/science/article/pii/S0304397598003582.

[10] J. Mayfield, T. W. Finin, and M. Hall. "Using automatic memoization as a software engineering tool in real-world AI systems". In: *Proceedings the 11th Conference on Artificial Intelligence for Applications* (1995), pp. 87–93.

[11] M. L. Scott. *Programming Language Pragmatics*. 3rd ed. Morgan Kaufmann, 2009. ISBN: 0123745144,9780123745149.

[12] P. Sestoft. *Programming Language Concepts*. Springer London, 2012, pp. 1–12.

[13] “The Free On-line Dictionary of Computing”. In: (2003). Normal order. URL: https://encyclopedia2.thefreedictionary.com/Normal+order.

[14] “The Free On-line Dictionary of Computing”. In: (2003). Applicative order. URL: https://encyclopedia2.thefreedictionary.com/Applicative+order.